THE SYMBOLIC PROGRAMMING ENVIRONMENT (SPE™)
A Common Lisp Development Environment For Sun Workstations

Aaron Endelman (endelman@sun.com)
Steve Gadol (sgadol@sun.com)
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

Why have a Specialized Common Lisp Programming Environment?

The constructs provided in the Common Lisp language provide a very powerful and flexible representation mechanism. The language alone, however, does not create an efficient work environment. Without tools that explicitly attack the problems of program organization, editing, and debugging, it remains a difficult task to exploit these advantages fully. What is needed is a collection of tools which, with knowledge about the relationship of the components in Lisp and a model of how Lisp programs are developed, encapsulates the tasks that constitute the Lisp programming process. Building such a programming environment in Lisp is a more tractable problem than in most conventional languages (such as C, FORTRAN, and Pascal) because the flexibility of Lisp allows programs to be operated on as data by other Lisp programs. This gives rise to the potential for a much tighter integration of tools than has typically been available.

A Symbolic Programming Environment

Sun's Symbolic Programming Environment (SPE™) is an example of just such an integration. Like the environments designed for dedicated hardware based implementations of Lisp, it is designed to be a functionally complete set of mechanisms for building Lisp programs. Many of the editing, data inspecting, and debugging features supplied in SPE are specializations of primitive capabilities implemented in the Sun Common Lisp product developed by Lucid, Inc. The windowing and object oriented programming mechanisms used in the product are derived from the HyperClass system created by Schlumberger.

SPE can be separated logically (albeit somewhat artificially) into two sets of tools: one for constructing and debugging procedures and data structures, and the other for building and maintaining entire applications.

Tools for the Procedure and Data Level

The construction and debugging of individual procedures, known as "programming-in-the-small", centers around the process of editing. In SPE the Editor is built in the style of Emacs. Like other Emacs implementations, it is built up from a set of character and buffer-managing primitives. And like other Emacs implementations it retains the flexibility of a dynamically changeable association between key strokes used to invoke commands and the procedures that implement them. The main difference between the SPE Editor and other UNIX™ Emacs implementations is that the Editor is written entirely in Common Lisp. New features that a programmer might wish to add to the environment are written in the same language and style as the applications the Editor is being used to develop. Further, it is straightforward to use features supplied by the Editor in application code that runs under SPE. Combined with the services supplied by the Lisp Window Tool Kit, the basic Editor modules were used to construct many of the tools contained in SPE.

To the SPE user the two most obvious uses of the Editor are the source file editor tool and Lisp Listener tool. The Listener acts like the UNIX™ shell in that it is the tool into which the user types commands to the Lisp system. Using the Editor to implement the Lisp Listener has several advantages over a simpler keyboard handler. The same set of keyboard and mouse commands is used to select and modify text. And all of the Editor's command extension facilities are available for use in defining new top level commands for the system. One example is the use of the Editor's parsing capability to isolate symbols in a Lisp form entered as a command. Coupled with Lisp's ability to retrieve properties of function symbols, it was a simple matter to create a command that translated the act of pointing at a partially completed command in the Lisp Listener into an operation that returned the sequence of expected parameters.

The SPE file editor tool utilizes the capability of the Editor modules as a full text editor, with both a fundamental text mode and a special Lisp mode. In the Lisp mode, the Editor has a variety of commands for operating on Lisp source code. The usual commands for indenting, balancing parentheses, and other functions are included, as are commands for evaluating and compiling individual functions, buffers, and files. A mouse sensitive command panel gives users quick access to several frequently executed commands.

Because the Editor is written in Lisp and running in the same address space as the rest of the Lisp system, the SPE Lisp Editor can directly map the services provided by SPE's Program Analyzer and Source Code Finder into a useful set of Lisp Mode commands that make it much easier for programmers to navigate through a complex application.

Another more substantial example of using the Editor modules was motivated by a significant change to the window tools used in SPE that occurred midway through the development project. Faced with the task of converting a very large body of SPE and Hyperclass system code and applications, the problem was solved by creating an interactive transformation program that read in and analyzed Lisp code using the window tools. Using a rule base that encapsulated syntactic and semantic knowledge about both the old and new forms of the window system, the transformation program formulated changes that were then presented to the programmer via the text managing and display mechanism in the SPE Editor. The use of the Editor simplified the task of manipulating the Lisp program text in a way that enabled the programmer to interactively examine and modify the transforms as needed.

SPE Program Analyzer

SPE's Program Analyzer is a Common Lisp program that builds a database about the structural relationships it discovers as it processes Lisp source code. The database produced by the Program Analyzer contains entries for defining forms, as well as notations that describe function calling and special variable-accessing patterns. This data can be used to answer a variety of questions about the static structure of a Lisp application. It is one of the mechanisms used by SPE to locate source code for a function given the symbol that names the function. It can also answer questions about which procedures a given named procedure calls. This information is used directly by an Editor command that lists the results, and recursive expansion of the query is used to create function call graphs

(Figure 1). Interactive tools of this kind have proven to be very valuable to programmers exploring the structure of a program they are not yet familiar with and in maintaining large complex programs.

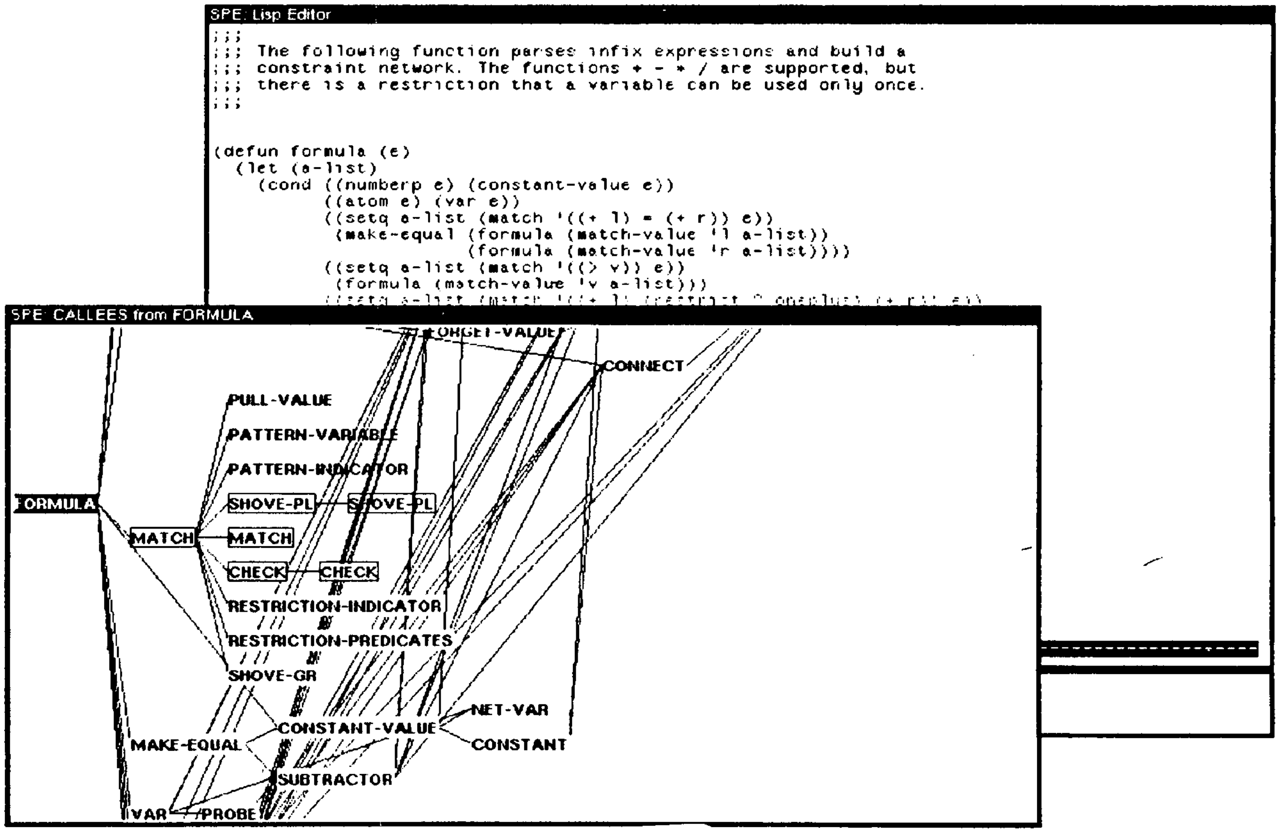
Operationally, the Program Analyzer is similar to the front end of the Lisp compiler. Like the compiler, it walks through the source it is processing, using its knowledge about the structure of the Common Lisp language. But rather than emitting machine code as the compiler does, it generates the database.

SPE Source Code Finder

Another important tool of SPE is the Source Code Finder. This tool is used to locate the defining source code for named objects, such as functions and structures. Given the symbol naming a Lisp object, the Source Code Finder attempts to locate the file in which the defining form is stored. It is not necessary that the source code be loaded into the Lisp system. This is important because in most applications, only the binaries produced by the Lisp compiler will be present once initial debugging on the application is completed. In addition to its usage in the Lisp Listener and Editor tools, this SPE component is used by the SPE debugger, function call graph manager, and by several of the specialized data object inspectors that have been created.

The Source Code Finder will use up to three different mechanisms to accomplish its task. The primary mechanism is a source-code recording facility supplied by the underlying

Figure 1. *An SPE Program Analyzer Call Graph. The Editor window underneath, containing the source code for the Lisp function* formula, *appeared when the* formula *node was selected with the mouse.*



```
;;;
;;; The following function parses infix expressions and build a
;;; constraint network. The functions + - * / are supported, but
;;; there is a restriction that a variable can be used only once.
;;;


(defun formula (e)
  (let (a-list)
    (cond ((numberp e) (constant-value e))
          ((atom e) (var e))
          ((setq a-list (match '((+ l) = (+ r)) e))
           (make-equal (formula (match-value 'l a-list))
                       (formula (match-value 'r a-list))))
          ((setq a-list (match '((> v)) e))
           (formula (match-value 'v a-list)))
```

II-2.7

Common Lisp. Here, the compiler and loader maintain a correspondence between Lisp objects and source files, so it is possible to determine the file containing source for any object entered into the Lisp system by the Lisp loader. As functions are recompiled and reloaded into the system, this mechanism automatically keeps the correspondence up to date. The second mechanism uses the database generated by the SPE Program Analyzer. Because this makes it possible to locate source code for objects not currently loaded into the system, it is quite useful when working on programs (including new versions of Lisp or SPE system code) that are not in a sufficiently consistent state to load them and work on them directly. The third makes it possible to use a tag file of the type produced by other UNIX^{TM} Emacs implementations.

Dynamic Debugging Tools

The tools described thus far center on the mechanisms for entering and modifying source. As such they are most concerned with the static or structural aspects of a Lisp application. Equally important to the productivity of the application developer is the ability to explore and manipulate the program's dynamic or runtime state. The SPE Data Inspector and SPE Debugger provide this basic capability.

These tools have their foundations in the Common Lisp system underlying SPE. The Lisp system is delivered with both an interactive inspector and a debugger, but the utility of these components is limited by the text oriented display and command mechanisms they use. SPE presents its data inspecting and debugging mechanisms to the programmer as tools fully integrated into the Lisp window system. Information needed to manipulate the dynamic program state is displayed in a tabular format which is refreshed as changes are made. Further, entries in the tables are mouse sensitive so the user can select items directly from the screen.

SPE Debugger and Data Inspector

The SPE Debugger is based on an abstraction in which the user's computation is viewed as a sequence of procedure calls, a history of which is kept on a push-down stack. The intent is to make it unnecessary for the programmer to deal with the structure of the Lisp implementation in order to debug application code. The user is able to view both the stack and the details of the procedure call frames it contains. Certain frames that correspond to activity within the underlying Lisp system can be optionally hidden from view both to condense the display and to reduce the complexity of the information presented.

The SPE Debugger tool comprises several subwindows, called *panes*. From left to right, top to bottom, the first pane is the *backtrace pane*, which presents a selectable, scrollable backtrace of all function frames up to the top level. To the right is the short *selected frame function pane*, which contains the name of the current function. Selecting that name with the mouse uses the Source Code Finder to bring up an Editor window containing the function's source code. Below this pane, the *selected frame parameter pane* shows the names and values of any arguments to the function, along with a label indicating their associated lambda-list keywords, e.g., REQUIRED, OPTIONAL. In addition, the values of the arguments displayed here can be changed or inspected in further detail with the mouse. The next pane shows the current error message.

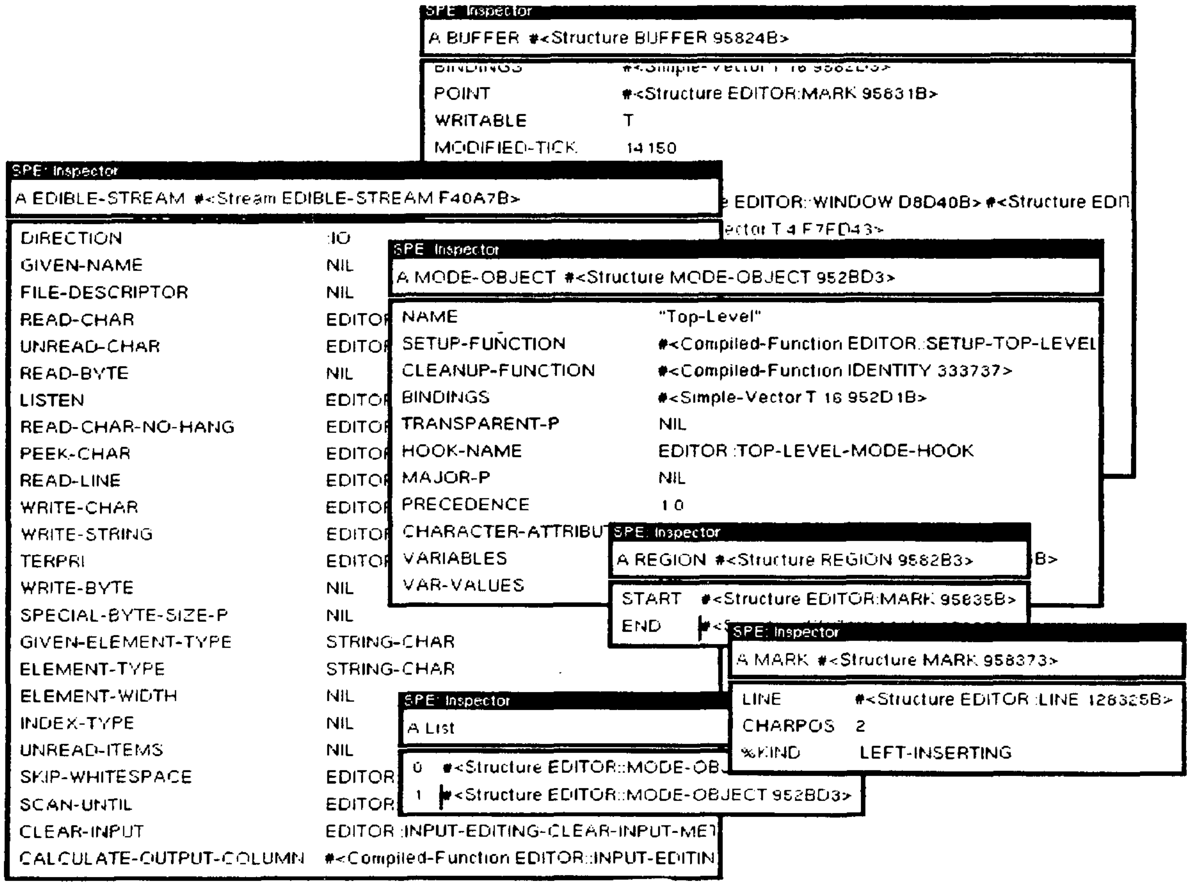
Finally, the bottommost pane contains eight command buttons. The most commonly used are ABORT, which stops the current computation and exits the Debugger; RETURN, which resumes the current computation by returning a user-supplied value from the current frame; RESTART, which tries to complete the computation by reinvoking the current frame; and CONTINUE, which resumes computation for continuable errors.

Clicking the mouse on a Debugger parameter value invokes the SPE Data Inspector. Data Inspectors display any arbitrary Lisp object (Figure 2). They are often invoked through the Debugger but may also be called programmatically through the Lisp *inspect* function. When the object being inspected is a structure, the individual slot names and values are shown. (For lists, arrays, and other unlabeled sequences, the slot values are enumerated instead.) Slot values may be both inspected or altered by selecting them with the mouse. Since this recursive nature of the Inspector can easily lead to the propagation of many Inspector windows, a menu (invoked with a click of the right mouse button) is supplied that allows all of the children and/or parents of a given Inspector window to be destroyed all at once.

Tools for the System Level

The tools discussed thus far, with the possible exception of the Program Analyzer, have been oriented around the kind of facilities a single programmer needs to create and debug modest size applications. Applications of AI technology, however, have tended to be large programs developed by several programmers working as a team. Thus it is also im-

Figure 2. *SPE Data Inspectors.*



```
SPE: Inspector
A BUFFER #<Structure BUFFER 95824B>
POINT                #<Structure EDITOR:MARK 95631B>
WRITABLE             T
MODIFIED-TICK        14150
                     EDITOR::WINDOW D8D40B> #<Structure EDI
```

```
SPE: Inspector
A EDIBLE-STREAM #<Stream EDIBLE-STREAM F40A7B>
DIRECTION                  :IO
GIVEN-NAME                 NIL
FILE-DESCRIPTOR            NIL
READ-CHAR                  EDITO
UNREAD-CHAR                EDITO
READ-BYTE                  NIL
LISTEN                     EDITO
READ-CHAR-NO-HANG          EDITO
PEEK-CHAR                  EDITO
READ-LINE                  EDITO
WRITE-CHAR                 EDITO
WRITE-STRING               EDITO
TERPRI                     EDITO
WRITE-BYTE                 NIL
SPECIAL-BYTE-SIZE-P        NIL
GIVEN-ELEMENT-TYPE         STRING-CHAR
ELEMENT-TYPE               STRING-CHAR
ELEMENT-WIDTH              NIL
INDEX-TYPE                 NIL
UNREAD-ITEMS               NIL
SKIP-WHITESPACE            EDITOR
SCAN-UNTIL                 EDITOR
CLEAR-INPUT                EDITOR :INPUT-EDITING-CLEAR-INPUT-ME
CALCULATE-OUTPUT-COLUMN    #<Compiled-Function EDITOR::INPUT-EDITIN
```

```
SPE: Inspector
A MODE-OBJECT #<Structure MODE-OBJECT 952BD3>
NAME                  "Top-Level"
SETUP-FUNCTION        #<Compiled-Function EDITOR::SETUP-TOP-LEVEL
CLEANUP-FUNCTION      #<Compiled-Function IDENTITY 333737>
BINDINGS              #<Simple-Vector T 16 952D1B>
TRANSPARENT-P         NIL
HOOK-NAME             EDITOR::TOP-LEVEL-MODE-HOOK
MAJOR-P               NIL
PRECEDENCE            1.0
CHARACTER-ATTRIBU
VARIABLES                                             B>
VAR-VALUES
```

```
SPE: Inspector
A REGION #<Structure REGION 9582B3>
START  #<Structure EDITOR:MARK 95635B>
END    #<
```

```
SPE: Inspector
A MARK #<Structure MARK 958373>
LINE      #<Structure EDITOR::LINE 128325B>
CHARPOS   2
%KIND     LEFT-INSERTING
```

```
SPE: Inspector
A List
0  #<Structure EDITOR::MODE-OB
1  #<Structure EDITOR::MODE-OBJECT 952BD3>
```

portant that the environment deal with productivity issues resulting from the complexity this dimension introduces.

The concept is known as "programming-in-the-large" and is concerned with the maintenance of collections of tens or hundreds of related files. At issue is the sequence of commands that must be executed to recompile, load and analyze pieces of an application when changes are made so that the application is brought back to a consistent state. Clearly, it is undesirable to be forced to recompile the entire application when small changes are made. The SPE Application Manager is designed to address this requirement.

In order to describe dependencies between application components, the Application Manager defines a linguistic extension to the Common Lisp notions of *module* and *require*. Modules are thought of as the collections of files that implement a specific capability. Because the underlying Lisp system has the capability to execute code written in other languages like C and FORTRAN, the Application Manager provides a mechanism that allows users to declare module types with specialized compilation and loading mechanisms. UNIX^{TM} file name type extensions are used to key module types.

Module definitions may also contain declared dependencies. These dependencies are other modules which the system must guarantee are up to date and loaded before any operations are perfomed. The Application Manager uses the transitive closure of these dependency declarations to build a plan for carrying out requests to load, compile or analyze. As the programmers make necessary changes to source files, they generally do not have to recompile the entire application or worry about sequence dependencies. In this sense the Application Manager serves a similar function to *make* in UNIX^{TM}.

Besides the basic linguistic and plan-generating mechanisms, the Application Manager contains a set of graphical tools to show the relationships between the components of the application and their status. This module graph display has proven particularly useful in that it acts as an interactive structural map of the application and supports a direct interface between the Application Manager and the SPE Editor.

Simplicity and Power

A strong adherence to a "keep-it-simple" philosophy is reflected throughout the SPE user-interface. The most frequently used SPE operations require the least work by programmers. A single mouse click is all that is required to compile a changed procedure definition. One mouse click, followed by a name, can load a file; a few more mouse clicks is all it takes to compile and load an entire application.

Using SPE, programmers can move easily from one tool to another within a particular dynamic context. For example, a programmer can invoke the Inspector on a data structure visible in the Debugger. Or the programmer can navigate from the Debugger to the source code of the errant procedure, fix the code in the Editor, and incrementally compile the changed definition. A programmer can display a call graph for a given top-level function, and then click on the name of any lower-level function shown to go to an Editor window containing its source code. SPE minimizes the steps and time taken through the edit-compile-debug cycle, thus maximizing the programmer's efficiency.

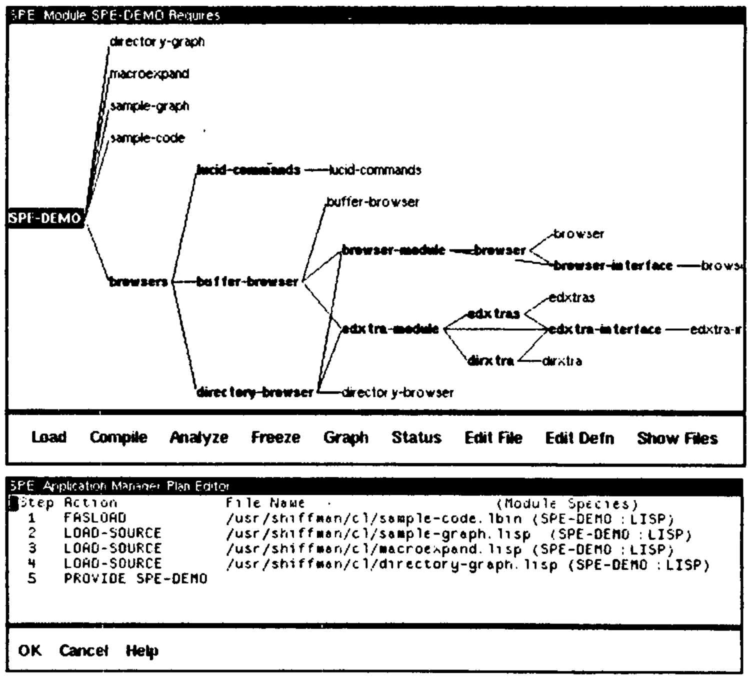
One important productivity lever for the long term growth of any programming environment comes from the idea that the environment can be turned back on itself and enhanced. By design the architecture of SPE encourages this kind of incremental development. The next section of this article describes how SPE is used to add a new utility to the SPE system.

Using SPE: an Example

Consider the following sample application which extends SPE. The application constructs a visual directory graph by using the predefined SPE browser/grapher facilities. This is the code necessary to define the structure of the application:

```
(DEFMODULES
  (:DIR "/usr/shiffman/cl"
  ;; Within this default directory...
    (:MODULE "SPE-DEMO"
    ;; ...we define the "SPE-DEMO" module...
      (:REQUIRE "browsers")
      ;; ...which requires the "browsers" module...
      "sample-code"
      "sample-graph"
      "macroexpand"
      "directory-graph"
      ;; ...and these four Lisp files to be loaded.
)))
```

These few lines are all the code the Application Manager needs. A more complex application might use multiple modules, contain non-Lisp code, or reside in multiple directories.



```
Step Action          File Name                          (Module Species)
  1   FASLOAD        /usr/shiffman/cl/sample-code.lbin (SPE-DEMO : LISP)
  2   LOAD-SOURCE    /usr/shiffman/cl/sample-graph.lisp  (SPE-DEMO : LISP)
  3   LOAD-SOURCE    /usr/shiffman/cl/macroexpand.lisp (SPE-DEMO : LISP)
  4   LOAD-SOURCE    /usr/shiffman/cl/directory-graph.lisp (SPE-DEMO : LISP)
  5   PROVIDE SPE-DEMO
```

Figure 3. *The SPE Application Manager module/file graph and Plan Editor.*

In those cases, the specifications needed are simple extensions to the basic syntax shown above.

Selecting the Lisp Listener's LOAD option instructs SPE to read in the file containing the module definition above. Now SPE understands the application's structure. Selecting the REQUIRES option lets you give SPE the name of the module you wish to load ("SPE-DEMO"). SPE now activates the Application Manager, and a visual graph appears (Figure 3), showing the application's module structure as you have defined it. Selecting the Application Manager's COMPILE option causes a plan to be built. This plan contains an ordered list of application files, noting which ones are out-of-date and need to be compiled, and which are up-to-date and can just have their binaries loaded. The Application Manager also notes whether each file is already loaded. If it was and the source file hasn't been modified, it will not attempt to reload or recompile the file.

The plan now appears in an Application Manager Plan Editor, which is just an SPE Editor which has been specialized for this task. Within the Plan Editor, you have the option of bypassing selected files if you so desire. Selecting OK in the Plan Editor runs the plan, compiling and loading files as necessary in the proper order.

The application implements a new Editor command called Show Directory Graph. It is a simple matter to invoke this command by typing a couple of keys followed by the name of the command itself. The Show Directory Graph command prompts for a directory which can then be entered.

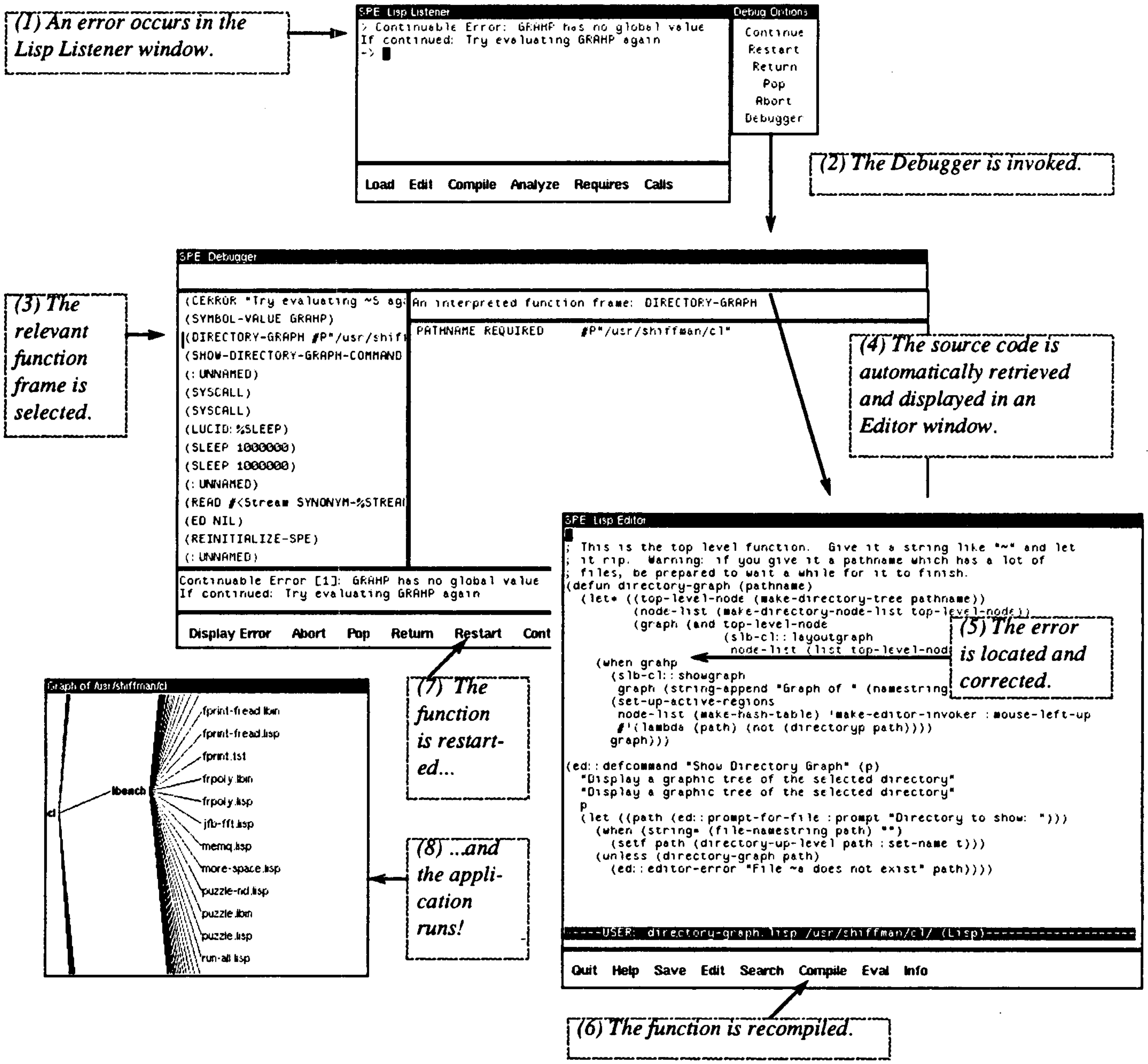
Unfortunately, there's a bug! Figure 4 shows the path taken to fix the bug, which will be described here in detail.

Upon encountering the bug, Lisp signals an error, and a menu appears. One of the menu options is to select the SPE Debugger. Using the Debugger to examine the broken function frame, note that the error's immediate cause was a failed attempt to evaluate the symbol *GRAHP*, which is probably just a typo of the word *GRAPH* somewhere in the source code. The function frame immediately prior to this one is that of the broken user-defined function. When this frame is selected, the function's name, *DIRECTORY-GRAPH*, is shown in the *selected frame function pane*. The name is mouse-sensitive, and selecting it starts up an Editor. The Source Code Finder locates the source file and the Editor reads it in, using information provided by the Finder to locate the function and scroll the buffer to its starting location. Examining the function visually reveals the typo, which is fixed by a few keystrokes. Selecting the Editor's COMPILE option recompiles the source directly into the Lisp environment. Now the Debugger's RESTART option is selected, and the function call is reinvoked, this time using the debugged function. Execution now proceeds without error, and a directory graph appears on the screen, as desired.

Future Directions

SPE is currently available as a programming environment for Sun Common Lisp. Refinements and additions to it continue, and it is being used at Sun as a development environment. Future versions of SPE will include specialized tools for the Common Lisp Object System (CLOS), the recently adopted extension to Common Lisp for object-oriented programming. Sun also plans a utility for visually stepping through source code while it is executing so that programmers can better follow the flow of control within a piece of assem-

Figure 4. *Interactive run-time debugging in SPE.*


(1) An error occurs in the Lisp Listener window.
SPE Lisp Listener
Debug Options
Continue
Restart
Return
Pop
Abort
Debugger
Load Edit Compile Analyze Requires Calls
(2) The Debugger is invoked.
SPE Debugger
(3) The relevant function frame is selected.
(4) The source code is automatically retrieved and displayed in an Editor window.
Display Error Abort Pop Return Restart Cont
SPE Lisp Editor
(5) The error is located and corrected.
Quit Help Save Edit Search Compile Eval Info
(6) The function is recompiled.
(7) The function is restart-ed...
(8) ...and the appli-cation runs!
Graph of /usr/shiffman/cl
cl
lbench
fprint-fread.lbin
fprint-fread.lisp
fprint.tst
frpoly.lbin
frpoly.lisp
jfb-fft.lisp
memq.lisp
more-space.lisp
puzzle-nd.lisp
puzzle.lbin
puzzle.lisp
run-all.lisp

```
> Continuable Error: GRAHP has no global value
If continued: Try evaluating GRAHP again
->
```

```
(CERROR "Try evaluating ~S ag
(SYMBOL-VALUE GRAHP)
(DIRECTORY-GRAPH #P"/usr/shif
(SHOW-DIRECTORY-GRAPH-COMMAND
(:UNNAMED)
(SYSCALL)
(SYSCALL)
(LUCID:%SLEEP)
(SLEEP 1000000)
(SLEEP 1000000)
(:UNNAMED)
(READ #<Stream SYNONYM-%STREA
(ED NIL)
(REINITIALIZE-SPE)
(:UNNAMED)
```

```
An interpreted function frame: DIRECTORY-GRAPH
PATHNAME REQUIRED     #P"/usr/shiffman/cl"
```

```
Continuable Error [1]: GRAHP has no global value
If continued: Try evaluating GRAHP again
```

```
; This is the top level function.  Give it a string like "~" and let
; it rip.  Warning: if you give it a pathname which has a lot of
; files, be prepared to wait a while for it to finish.
(defun directory-graph (pathname)
  (let* ((top-level-node (make-directory-tree pathname))
         (node-list (make-directory-node-list top-level-node))
         (graph (and top-level-node
                     (slb-cl::layoutgraph
                      node-list (list top-level-nod
    (when grahp
      (slb-cl::showgraph
       graph (string-append "Graph of " (namestring
      (set-up-active-regions
       node-list (make-hash-table) 'make-editor-invoker :mouse-left-up
       #'(lambda (path) (not (directoryp path))))
      graph)))

(ed::defcommand "Show Directory Graph" (p)
  "Display a graphic tree of the selected directory"
  "Display a graphic tree of the selected directory"
  p
  (let ((path (ed::prompt-for-file :prompt "Directory to show: ")))
    (when (string= (file-namestring path) "")
      (setf path (directory-up-level path :set-name t)))
    (unless (directory-graph path)
      (ed::editor-error "File ~a does not exist" path))))
```

```
-----USER: directory-graph.lisp /usr/shiffman/cl/ (Lisp)-------------------
```

bled Lisp code. Other utilities are planned that will make SPE more productive for Lisp and object-oriented programming.

Within Sun, SPE serves as a host for continued exploration of programming environments and better productivity tools for advanced application development.

Acknowledgements

This article is condensed from another which appeared in the Winter 1989 issue of Sun Technology. The authors would like to thank Sun Technology for permission to reprint portions of that article here.

References

D. R. Barstow, H. E. Shrobe, E. Sandewall. Interactive Programming Environments. New York: McGraw-Hill, 1984.

S. Gadol. "SPE -- A Common Lisp Environment on Workstations." Proceedings of the Fourth Annual Artificial Intelligence & Advanced Computer Technology Conference. Glen Ellyn, IL: Tower Conference Management Company, 1988, pp. 167-176.

R. G. Smith, R. Dinitz, P. Barth. "Impulse-86: A Substrate for Object-Oriented Interface Design." OOPSLA '86 Conference Proceedings. New York: Association for Computing Machinery, 1986, pp. 399-404.

SPE User's Guide. Mountain View, CA: Sun Microsystems, Inc., 1987.

G. Steele Jr. Common LISP: The Language. Burlington, Mass.: Digital Press, 1984.

Sun Common Lisp User's Guide. Mountain View, CA: Sun Microsystems, Inc., 1986.